What is Spring Boot

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

It is a Spring module that provides the **RAD (*Rapid Application Development*)** feature to the Spring Framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.



In short, Spring Boot is the combination of **Spring Framework** and **Embedded Servers**.

In Spring Boot, there is no requirement for XML configuration (deployment descriptor). It uses convention over configuration software design paradigm that means it decreases the effort of the developer.

We can use Spring **STS IDE** or **Spring Initializer** to develop Spring Boot Java applications.

**Why should we use Spring Boot Framework?**

We should use Spring Boot Framework because:

* The dependency injection approach is used in Spring Boot.
* It contains powerful database transaction management capabilities.
* It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc.
* It reduces the cost and development time of the application.

Along with the Spring Boot Framework, many other Spring sister projects help to build applications addressing modern business needs. There are the following Spring sister projects are as follows:

* **Spring Data:** It simplifies data access from the relational and **NoSQL** databases.
* **Spring Batch:** It provides powerful **batch** processing.
* **Spring Security:** It is a security framework that provides robust **security** to applications.
* **Spring Social:** It supports integration with **social networking** like LinkedIn.
* **Spring Integration:** It is an implementation of Enterprise Integration Patterns. It facilitates integration with other **enterprise applications** using lightweight messaging and declarative adapters.

Advantages of Spring Boot

* It creates **stand-alone** Spring applications that can be started using Java **-jar**.
* It tests web applications easily with the help of different **Embedded** HTTP servers such as **Tomcat, Jetty,** etc. We don't need to deploy WAR files.
* It provides opinionated '**starter**' POMs to simplify our Maven configuration.
* It provides **production-ready** features such as **metrics, health checks,** and **externalized configuration**.
* There is no requirement for **XML** configuration.
* It offers a **CLI** tool for developing and testing the Spring Boot application.
* It offers the number of **plug-ins**.
* It also minimizes writing multiple **boilerplate codes** (the code that has to be included in many places with little or no alteration), XML configuration, and annotations.
* It **increases productivity** and reduces development time.

Limitations of Spring Boot

Spring Boot can use dependencies that are not going to be used in the application. These dependencies increase the size of the application.

Goals of Spring Boot

The main goal of Spring Boot is to reduce **development, unit test,** and **integration test** time.

* Provides Opinionated Development approach
* Avoids defining more Annotation Configuration
* Avoids writing lots of import statements
* Avoids XML Configuration.

By providing or avoiding the above points, Spring Boot Framework reduces **Development time, Developer Effort,** and **increases productivity**.

Prerequisite of Spring Boot

To create a Spring Boot application, following are the prerequisites. In this tutorial, we will use **Spring Tool Suite** (STS) IDE.

* Java 1.8
* Maven 3.0+
* Spring Framework 5.0.0.BUILD-SNAPSHOT
* An IDE (Spring Tool Suite) is recommended.

Spring Boot Features

* Web Development
* SpringApplication
* Application events and listeners
* Admin features
* Externalized Configuration
* Properties Files
* YAML Support
* Type-safe Configuration
* Logging
* Security

**Web Development**

It is a well-suited Spring module for web application development. We can easily create a self-contained HTTP application that uses embedded servers like **Tomcat, Jetty,** or Undertow. We can use the **spring-boot-starter-web** module to start and run the application quickly.

**SpringApplication**

The SpringApplication is a class that provides a convenient way to bootstrap a Spring application. It can be started from the main method. We can call the application just by calling a static run() method.

1. **public** **static** **void** main(String[] args)
2. {
3. SpringApplication.run(ClassName.**class**, args);
4. }

**Application Events and Listeners**

Spring Boot uses events to handle the variety of tasks. It allows us to create factories file that is used to add listeners. We can refer it to using the **ApplicationListener key**.

Always create factories file in META-INF folder like **META-INF/spring.factories**.

**Admin Support**

Spring Boot provides the facility to enable admin-related features for the application. It is used to access and manage applications remotely. We can enable it in the Spring Boot application by using **spring.application.admin.enabled** property.

**Externalized Configuration**

Spring Boot allows us to externalize our configuration so that we can work with the same application in different environments. The application uses YAML files to externalize configuration.

**Properties Files**

Spring Boot provides a rich set of **Application Properties**. So, we can use that in the properties file of our project. The properties file is used to set properties like **server-port =8082** and many others. It helps to organize application properties.

**YAML Support**

It provides a convenient way of specifying the hierarchical configuration. It is a superset of JSON. The SpringApplication class automatically supports YAML. It is an alternative of properties file.

**Type-safe Configuration**

The strong type-safe configuration is provided to govern and validate the configuration of the application. Application configuration is always a crucial task which should be type-safe. We can also use annotation provided by this library.

**Logging**

Spring Boot uses Common logging for all internal logging. Logging dependencies are managed by default. We should not change logging dependencies if no customization is needed.

**Security**

Spring Boot applications are spring bases web applications. So, it is secure by default with basic authentication on all HTTP endpoints. A rich set of Endpoints is available to develop a secure Spring Boot application.

# **Spring Boot Version**

The latest version of Spring Boot is **2.0**. It introduces a lot of new features along with some modifications and replacement.

## Spring Boot 2.0

Let's have a sneak peek at Spring Boot 2.0.

* **What's New**  
            Infrastructure Upgrade  
            Spring Framework 5
* **What's Changed**  
            Configuration Properties  
            Gradle Plugin  
            Actuators endpoints
* **What's Evolving**  
            Security  
            Metrics

The pivotal team has upgraded the **infrastructure** in which the following tools are involved:

* Supports **Java 8** or above versions
* Supports Apache **Tomcat 8** or above versions
* Supports **Thymeleaf 3**
* Supports **Hibernate 5.2**

In **Spring Framework 5**, the Pivotal team upgraded the following:

* Reactive Spring
  1. Servlet stack
     1. Servlet Container
     2. Servlet API
     3. Spring MVC
  2. **Reactive Stack**
     1. Netty, Servlet 3.1, Undertow
     2. Reactive HTTP Layer
     3. Spring WebFlux
* Functional API
* Kotlin Support

The latest version of Spring Boot is 2.2.1. This release of Spring Boot includes 110 fixes, dependency upgrades, and improvements.

In the Spring Boot v2.2.1, the annotation **@ConfigurationProperties** scanning is now disabled by default. We need to be explicitly opted into by adding the **@ConfigurationPropertiesScan** annotation.

## New Features

* Support constructor binding for property nested inside a JavaBean
* Add config property for CodecConfigurer.maxInMemorySize in WebFlux
* Make test slices' type exclude filters public
* Support amqps:// URIs in spring.rabbitmq.addresses

## Dependency upgrades

Some dependencies have been upgraded in Spring Boot v2.2.1 are as follows:

* Mongodb 3.11.2
* Spring Security 5.2.1.RELEASE
* Slf4j 1.7.29
* Spring Hateoas 1.0.1.RELEASE
* Hibernate Validator 6.0.18.Final
* Hibernate 5.4.8.Final
* Jetty 9.4.22.v20191022
* Spring Framework 5.2.1
* Spring AMQP 2.2.1
* H2 1.4.200
* Spring Security 5.2
* Spring Batch 4.2

Some important and widely used third-party dependencies are upgraded in this release are as follows:

* Micrometer 1.3.1
* Flyway 6.0.7
* Elasticsearch 6.8.4
* JUnit 5.5
* Jackson 2.10

## Performance Improvements

In Spring Boot 2.2.1 the following performance has been improved:

**Lazy Initialization**

In Spring Boot 2.2.1, we can enable global lazy initialization by using the property **spring.main.lazy-initialization** property. It reduces the application startup time.

**Java 13 Support**

Spring Boot 2.2.1 now supports the latest version of Java that is Java 13.

**Immutable Binding**

In the newer version of Spring Boot, Configuration properties support constructor-based binding. The class annotates with**@ConfigurationProperties**annotation is to be immutable. It can be enabled by adding an annotation **@ConfugurationProperties** to a class or one of its constructors with **@ConstructorBinding.**

**RSocket Support**

It is a part of **Spring Security**. RSocket integration is auto-configured when an application finds **spring-security-rsocket** is present on the classpath.

## Deprecations in Spring Boot 2.2

* The property **logging.file** has renamed to logging.file.name.
* The property **logging.path** has renamed to logging.file.path.
* The server.connection-timeout property has been deprecated in favor of server-specific properties.
* Joda time support is deprecated in favor of java.time.

**The following improvements are made in the Spring Boot 2.2.1**

* **Java:** Spring Boot 2.2.1
* **Spring Framework 5.2:** This release of Spring Boot upgrades to Spring Framework to 5.2.
* **JMX is disabled:** In this version, JMX is not enabled by default. We can enable it by using the property **jmx.enabled=true**. If you are using the IDE feature to monitor your application, we need to enable it.
* **Fork enabled by default:** Spring Boot application that ran by Maven Plugin is now forked by default.
* **JUnit 5:** Spring Boot v2.2.1 provides **JUnit 5**by default. JUnit 5's vintage engine is also included by default that supports existing JUnit 4-based test classes. We can also use JUnit 4 and JUnit 5 based test classes in the same module.
* **AssertJ 3.12:** This release of Spring Boot upgrades to AssertJ 3.12. It contains a breaking API changes for assertions related to Iterator.
* **Hibernate Dialect:** In the newer version of Spring Boot, Hibernate chose the dialect to use rather than applying a default dialect based on the detected database.
* **Gradle Requirements:** The latest version of Spring Boot requires Gradle 4.10.

## Spring Boot vs. Spring MVC

**Spring Boot:** Spring Boot makes it easy to quickly bootstrap and start developing a Spring-based application. It avoids a lot of boilerplate code. It hides a lot of complexity behind the scene so that the developer can quickly get started and develop Spring-based applications easily.

**Spring MVC:** Spring MVC is a Web MVC Framework for building web applications. It contains a lot of configuration files for various capabilities. It is an HTTP oriented web application development framework.

Ex:

Pom.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.4.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.example</groupId>

<artifactId>demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>demo</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Alien.java

-----------

package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

public class Alien {

private int aid;

private String aname;

private String tech;

@Autowired

@Qualifier("lap1")

private Laptop laptop;

public Alien() {

super();

System.out.println("object created..");

}

public int getAid() {

return aid;

}

public void setAid(int aid) {

this.aid = aid;

}

public String getAname() {

return aname;

}

public void setAname(String aname) {

this.aname = aname;

}

public String getTech() {

return tech;

}

public void setTech(String tech) {

this.tech = tech;

}

public Laptop getLaptop() {

return laptop;

}

public void setLaptop(Laptop laptop) {

this.laptop = laptop;

}

public void show()

{

System.out.println("in show");

laptop.compile();

}

}

Laptop.java

----------------

package com.example.demo;

import org.springframework.stereotype.Component;

@Component("lap1")

public class Laptop

{

private int lid;

private String brand;

public int getLid() {

return lid;

}

public void setLid(int lid) {

this.lid = lid;

}

public String getBrand() {

return brand;

}

public void setBrand(String brand) {

this.brand = brand;

}

@Override

public String toString() {

return "Laptop [lid=" + lid + ", brand=" + brand + "]";

}

public void compile() {

System.out.println("compiling");

}

}

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

ConfigurableApplicationContext context = SpringApplication.run(DemoApplication.class, args);

Alien a = context.getBean(Alien.class);

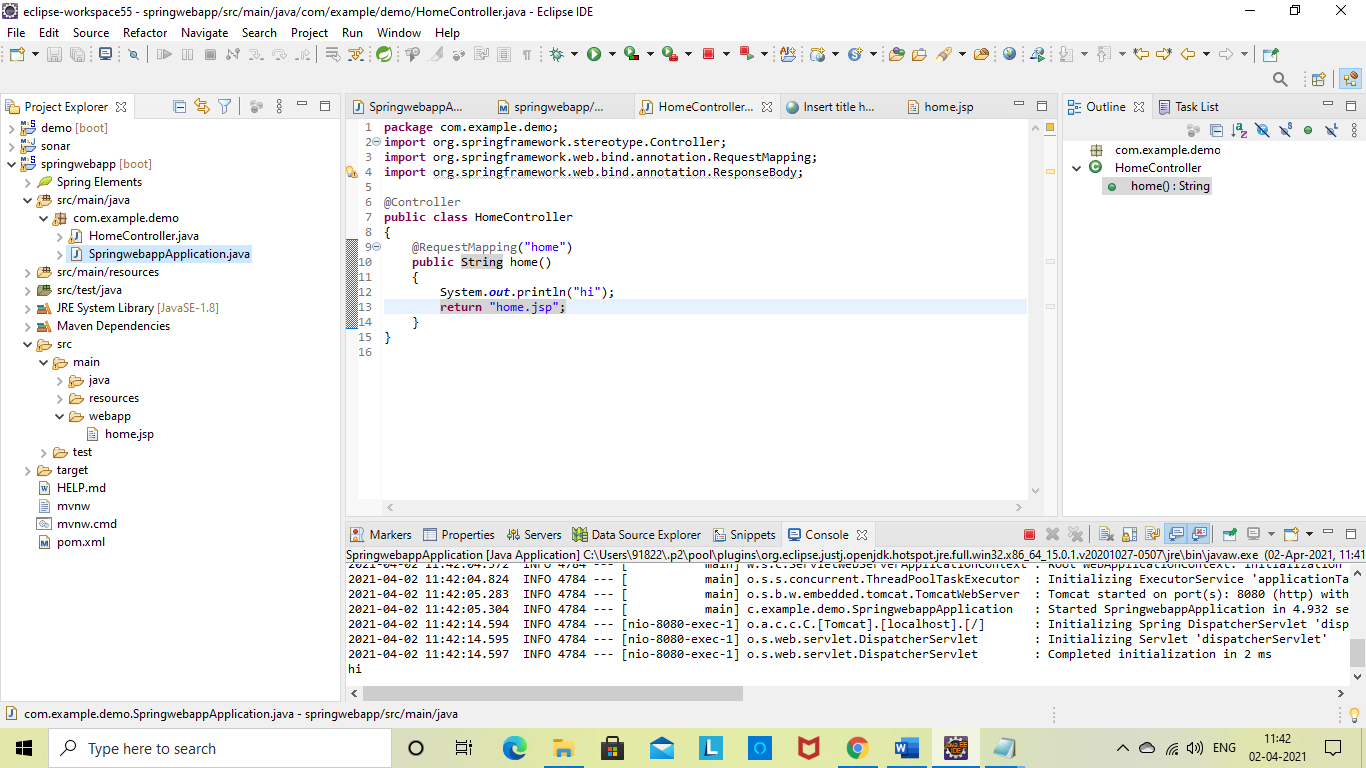
a.show();

}

}

Ex2

MyWebApp



<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.4.4</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.example</groupId>

<artifactId>springwebapp</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>springwebapp</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-jasper</artifactId>

<version>9.0.22</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

**package** com.example.demo;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.ResponseBody;

@Controller

**public** **class** HomeController

{

@RequestMapping("home")

**public** String home()

{

System.***out***.println("hi");

**return** "home.jsp";

}

}

Home.jsp

--------------

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

I love Java

</body>

</html>

Ex3:

Add below data in application.properties and create one sub folder inside webapp-----pages and keep home.jsp in the pages folder

spring.mvc.view.prefix=/pages/

spring.mvc.view.suffix=.jsp

WebApp using SpringBoot Accepting Client Data

Ex4:

**package** com.example.demo;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.servlet.ModelAndView;

@Controller

**public** **class** HomeController

{

@RequestMapping("home")

**public** ModelAndView home(@RequestParam("name") String myName)

{

System.***out***.println("hi");

ModelAndView mv=**new** ModelAndView();

mv.addObject("name", myName);

mv.setViewName("home");

**return** mv;

}

}

Home.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"* pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

welcome ${name}

</body>

</html>

WebApp using SpringBoot Model Object

Ex5:

**package** com.example.demo;

**import** org.springframework.stereotype.Component;

**public** **class** Alien {

**private** **int** aid;

**private** String aname;

**private** String tech;

//setters/getters

}

**package** com.example.demo;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.servlet.ModelAndView;

@Controller

**public** **class** HomeController

{

@RequestMapping("home")

**public** ModelAndView home(Alien alien)

{

System.***out***.println("hi");

ModelAndView mv=**new** ModelAndView();

mv.addObject("obj", alien);

mv.setViewName("home");

**return** mv;

}

}

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"* pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

welcome ${obj.aid} ,${obj.aname}, ${obj.tech}

</body>

</html>

URL: <http://localhost:8080/home?aid=10&aname=vijay&tech=java>

# **Spring Boot Annotations**

Spring Boot Annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide **supplemental** information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

Core Spring Framework Annotations

**@Required:** It applies to the **bean** setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception **BeanInitilizationException**.

**Example**

**public** **class** Machine

{

**private** Integer cost;

@Required

**public** **void** setCost(Integer cost)

{

**this**.cost = cost;

}

**public** Integer getCost()

{

**return** cost;

}

}

**@Autowired:** Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

**Example**

@Component

**public** **class** Customer

{

**private** Person person;

@Autowired

**public** Customer(Person person)

{

**this**.person=person;

}

}

**@Configuration:** It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

**Example**

@Configuration

**public** **class** Vehicle

{

@Bean

Vehicle engine()

{

**return** **new** Vehicle();

}

}

**@ComponentScan:** It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

**Example**

@ComponentScan(basePackages = "com.java")

@Configuration

**public** **class** ScanComponent

{

// ...

}

**@Bean:** It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

**Example**

@Bean

**public** BeanExample beanExample()

{

**return** **new** BeanExample ();

}

## Spring Framework Stereotype Annotations

**@Component:** It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with **@Component** is found during the classpath. The Spring Framework pick it up and configure it in the application context as a **Spring Bean**.

**Example**

@Component

**public** **class** Student

{

.......

}

**@Controller:** The @Controller is a class-level annotation. It is a specialization of **@Component**. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.

**Example**

@Controller

@RequestMapping("books")

**public** **class** BooksController

{

@RequestMapping(value = "/{name}", method = RequestMethod.GET)

**public** Employee getBooksByName()

{

**return** booksTemplate;

}

}

**@Service:** It is also used at class level. It tells the Spring that class contains the **business logic**.

**Example**

**package** com.java;

@Service

**public** **class** TestService

{

**public** **void** service1()

{

//business code

}

}

**@Repository:** It is a class-level annotation. The repository is a **DAOs** (Data Access Object) that access the database directly. The repository does all the operations related to the database.

**package** com.java;

@Repository

**public** **class** TestRepository

{

**public** **void** delete()

{

//persistence code

}

}

## Spring Boot Annotations

* **@EnableAutoConfiguration:** It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. **@SpringBootApplication**.
* **@SpringBootApplication:** It is a combination of three annotations **@EnableAutoConfiguration, @ComponentScan,** and **@Configuration**.

### **Spring MVC and REST Annotations**

* **@RequestMapping:** It is used to map the **web requests**. It has many optional elements like **consumes, header, method, name, params, path, produces**, and **value**. We use it with the class as well as the method.

**Example**

@Controller

**public** **class** BooksController

{

@RequestMapping("/computer-science/books")

**public** String getAllBooks(Model model)

{

//application code

**return** "bookList";

}

* **@GetMapping:** It maps the **HTTP GET** requests on the specific handler method. It is used to create a web service endpoint that **fetches** It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**
* **@PostMapping:** It maps the **HTTP POST**requests on the specific handler method. It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**
* **@PutMapping:** It maps the **HTTP PUT** requests on the specific handler method. It is used to create a web service endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**
* **@DeleteMapping:** It maps the **HTTP DELETE** requests on the specific handler method. It is used to create a web service endpoint that **deletes**a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**
* **@PatchMapping:** It maps the **HTTP PATCH**requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**
* **@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP MessageConverters** to convert the body of the request. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.
* **@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.
* **@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.
* **@RequestParam:** It is used to extract the query parameters from the URL. It is also known as a **query parameter**. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.
* **@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.**For each detail in the header, we should specify separate annotations. We can use it multiple time in a method
* **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody**annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.
* **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

Exception Handling

Handling exceptions and errors in APIs and sending the proper response to the client is good for enterprise applications.

Before proceeding with exception handling, let us gain an understanding on the following annotations.

## Controller Advice

The @ControllerAdvice is an annotation, to handle the exceptions globally.

## Exception Handler

The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client.

You can use the following code to create @ControllerAdvice class to handle the exceptions globally −

package com.java.demo.exception;

import org.springframework.web.bind.annotation.ControllerAdvice;

@ControllerAdvice

public class ProductExceptionController {

}

Define a class that extends the RuntimeException class.

package com.java.demo.exception;

public class ProductNotfoundException extends RuntimeException {

private static final long serialVersionUID = 1L;

}

You can define the @ExceptionHandler method to handle the exceptions as shown. This method should be used for writing the Controller Advice class file.

@ExceptionHandler(value = ProductNotfoundException.class)

public ResponseEntity<Object> exception(ProductNotfoundException exception) {

}

Now, use the code given below to throw the exception from the API.

@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)

public ResponseEntity<Object> updateProduct() {

throw new ProductNotfoundException();

}

The complete code to handle the exception is given below. In this example, we used the PUT API to update the product. Here, while updating the product, if the product is not found, then return the response error message as “Product not found”. Note that the **ProductNotFoundException** exception class should extend the **RuntimeException**.

package com.java.demo.exception;

public class ProductNotfoundException extends RuntimeException {

private static final long serialVersionUID = 1L;

}

The Controller Advice class to handle the exception globally is given below. We can define any Exception Handler methods in this class file.

package com.java.demo.exception;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice

public class ProductExceptionController {

@ExceptionHandler(value = ProductNotfoundException.class)

public ResponseEntity<Object> exception(ProductNotfoundException exception) {

return new ResponseEntity<>("Product not found", HttpStatus.NOT\_FOUND);

}

}

The Product Service API controller file is given below to update the Product. If the Product is not found, then it throws the **ProductNotFoundException** class.

package com.java.demo.controller;

import java.util.HashMap;

import java.util.Map;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RestController;

import com.java.demo.exception.ProductNotfoundException;

import com.java.demo.model.Product;

@RestController

public class ProductServiceController {

private static Map<String, Product> productRepo = new HashMap<>();

static {

Product honey = new Product();

honey.setId("1");

honey.setName("Honey");

productRepo.put(honey.getId(), honey);

Product almond = new Product();

almond.setId("2");

almond.setName("Almond");

productRepo.put(almond.getId(), almond);

}

@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)

public ResponseEntity<Object> updateProduct(@PathVariable("id") String id, @RequestBody Product product) {

if(!productRepo.containsKey(id))throw new ProductNotfoundException();

productRepo.remove(id);

product.setId(id);

productRepo.put(id, product);

return new ResponseEntity<>("Product is updated successfully", HttpStatus.OK);

}

}

The code for main Spring Boot application class file is given below −

package com.java.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

The code for **POJO class** for Product is given below −

package com.java.demo.model;

public class Product {

private String id;

private String name;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

The code for **Maven build – pom.xml** is shown below −

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>